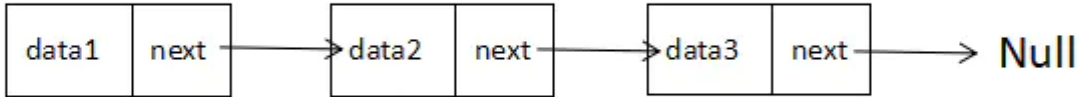


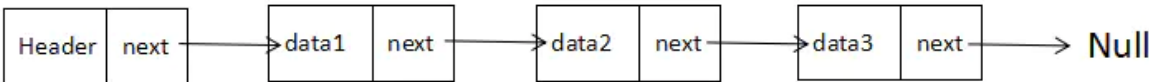
链表的定义

首先，要实现链表，我们先搞懂一些链表的基本东西，因为这很重要！
链表是一组节点组成的集合，每个节点都使用一个对象的引用来指向它的后一个节点。指向另一节点的引用讲做链。下面我画了一个简单的链接结构图，方便大家理解。



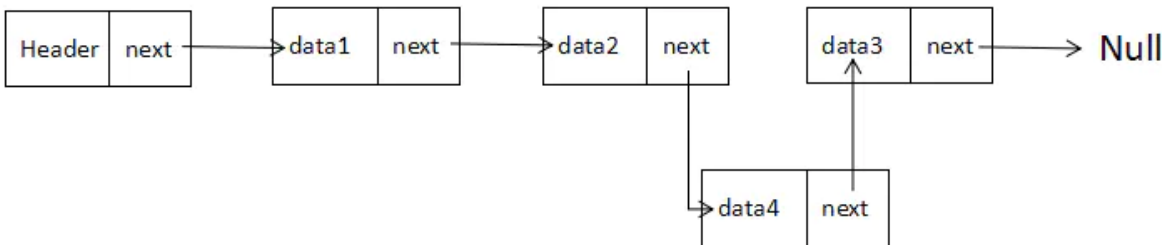
链表结构图

其中，data中保存着数据，next保存着下一个链表的引用。上图中，我们说 data2 跟在 data1 后面，而不是说 data2 是链表中的第二个元素。上图，值得注意的是，我们将链表的尾元素指向了 null 节点，表示链接结束的位置。
由于链表的起始点的确定比较麻烦，因此很多链表的实现都会在链表的最前面添加一个特殊的节点，称为 **头节点**，表示链表的头部。进过改造，链表就成了如下的样子：



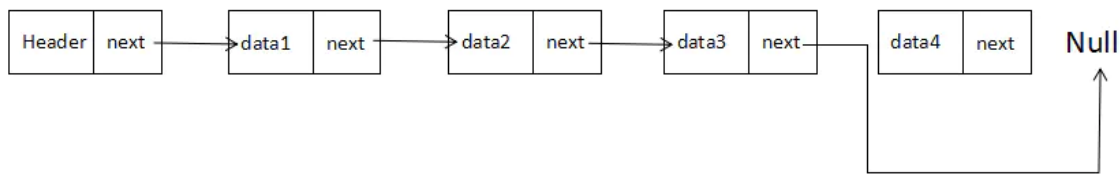
有头节点的链表

向链表中**插入一个节点**的效率很高，需要修改它前面的节点(前驱)，使其指向新加入的节点，而将新节点指向原来前驱节点指向的节点即可。下面我将用图片演示如何在 data2 节点 后面插入 data4 节点。



插入节点

同样，从链表中删除一个节点，也很简单。只需将待删节点的前驱节点指向待删节点的，同时将待删节点指向null，那么节点就删除成功了。下面我们用图片演示如何从链表中删除 data4 节点。



删除节点

链表的设计

我们设计链表包含两个类，一个是 Node 类用来表示节点，另一个是 LinkedList 类提供插入节点、删除节点等一些操作。

Node类

Node类包含两个属性：element 用来保存节点上的数据，next 用来保存指向下一个节点的链接，具体实现如下：

```
1 //节点
2 function Node(element) {
3     this.element = element;    //当前节点的元素
4     this.next = null;          //下一个节点链接
5 }
```

LinkedList类

LinkedList类提供了对链表进行操作的方法，包括插入删除节点，查找给定的值等。值得注意的是，它只有一个

属性，那就是使用一个 Node 对象来保存该链表的头节点。

它的构造函数的实现如下：

```
1 //链表类
2 function LList () {
3     this.head = new Node( 'head' );    //头节点
```

```

4     this.find = find;           //查找节点
5     this.insert = insert;       //插入节点
6     this.remove = remove;      //删除节点
7     this.findPrev = findPrev;   //查找前一个节点
8     this.display = display;     //显示链表
9 }

```

head节点的next属性初始化为 null，当有新元素插入时，next会指向新的元素。

接下来，我们来看看具体方法的实现。

insert：向链表插入一个节点

我们先分析分析insert方法，想要插入一个节点，我们必须明确要在哪个节点的前面或后面插入。我们先来看看，如何在一个已知节点的后面插入一个节点。

在一个已知节点后插入新节点，我们首先得找到该节点，为此，我们需要一个 find 方法用来遍历链表，查找给定的数据。如果找到，该方法就返回保存该数据的节点。那么，我们先实现 find 方法。

find：查找给定节点

```

1 //查找给定节点
2
3 function find ( item ) {
4     var currNode = this.head;
5     while ( currNode.element != item ){
6         currNode = currNode.next;
7     }
8     return currNode;
9 }

```

find 方法同时展示了如何在链表上移动。首先，创建一个新节点，将链表的头节点赋给这个新创建的节点，然后在链表上循环，如果当前节点的 element 属性和我们要找的信息不符，就将当前节点移动到下一个节点，如果查找成功，该方法返回包含该数据的节点；否则，就会返回null。

一旦找到了节点，我们就可以将新的节点插入到链表中了，将新节点的 next 属性设置为后面节点的 next 属性对应的值，然后设置后面节点的 next 属性指向新的节点，具体实现如下：

```

1 //插入节点
2 function insert ( newElement , item ) {

```

```

3     var newNode = new Node( newElement );
4     var currNode = this.find( item );
5     newNode.next = currNode.next;
6     currNode.next = newNode;
7 }

```

现在我们可以测试我们的链表了。等等，我们先来定义一个 display 方法显示链表的元素，不然我们怎么知道对不对呢？

display: 显示链表

```

1 //显示链表元素
2
3 function display () {
4     var currNode = this.head;
5     while ( !(currNode.next == null) ){
6         console.log( currNode.next.element );
7         currNode = currNode.next;
8     }
9 }

```

实现原理同上，将头节点赋给一个新的变量，然后循环链表，直到当前节点的 next 属性为 null 时停止循环，我们循环过程中将每个节点的数据打印出来就好了。

```

1 var fruits = new LList();
2
3 fruits.insert('Apple' , 'head');
4 fruits.insert('Banana' , 'Apple');
5 fruits.insert('Pear' , 'Banana');
6
7 console.log(fruits.display());      // Apple
8                                     // Banana
9                                     // Pear

```

remove: 从链表中删除一个节点

1 从链表中删除节点时，我们先要找个待删除节点的前一个节点，找到后，我们修改它的 `next` 属性，
2 使其不再指向待删除的节点，而是待删除节点的下一个节点。那么，我们就得需要定义一个 `findPrevious`
3 方法遍历链表，检查每一个节点的下一个节点是否存储待删除的数据。如果找到，返回该节点，这样就
4 可以修改它的 `next` 属性了。 `findPrevious` 的实现如下：

```
5  
6 //查找带删除节点的前一个节点  
7  
8 function findPrev( item ) {  
9     var currNode = this.head;  
10    while ( !( currNode.next == null ) && ( currNode.next.element != item ) ){  
11        currNode = currNode.next;  
12    }  
13    return currNode;  
14 }
```

这样，`remove` 方法的实现也就迎刃而解了

```
1 //删除节点  
2  
3 function remove ( item ) {  
4     var prevNode = this.findPrev( item );  
5     if( !( prevNode.next == null ) ){  
6         prevNode.next = prevNode.next.next;  
7     }  
8 }
```

我们接着写一段测试程序，测试一下 `remove` 方法：

```
1 // 接着上面的代码，我们再添加一个水果  
2  
3 fruits.insert('Grape' , 'Pear');  
4 console.log(fruits.display()); // Apple
```

```

5                                     // Banana
6                                     // Pear
7                                     // Grape
8
9 // 我们把香蕉吃掉
10
11 fruits.remove('Banana');
12 console.log(fruits.display());      // Apple
13                                     // Pear
14                                     // Grape

```

Great! 成功了，现在你已经可以实现一个基本的单向链表了。

```

1  <script>
2    // 节点
3    function Node(element) {
4      this.element = element;
5      this.next = null;
6    };
7    // 链表
8    function List() {
9      this.head = new Node('head'); //头节点
10     this.find = find;             //查找节点
11     this.insert = insert;         //插入节点
12     this.remove = remove;         //删除节点
13     this.findPrev = findPrev;     //查找前一个节点
14     this.display = display;       //显示链表
15   }
16
17   // find: 查找给定节点
18   function find(item) {
19     let currentNode = this.head;
20     while (currentNode.element !== item) {
21       currentNode = currentNode.next;
22     }
23     return currentNode;
24   };
25
26   // 插入节点
27   function insert(newElement, item) {
28     let newNode = new Node(newElement);
29     let currentNode = this.find(item);
30     newNode.next = currentNode.next;
31     currentNode.next = newNode;
32   };
33
34   // 显示链表
35   function display() {
36     let currentNode = this.head;
37     while (!(currentNode.next == null)) {
38       console.log(currentNode.next.element);
39       currentNode = currentNode.next;
40     }
41   };
42   // eg:

```

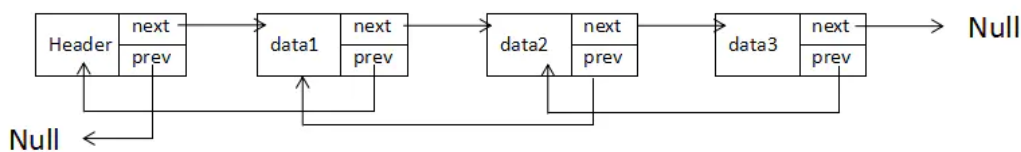
```

42 // eg.
43 var fruits = new List();
44
45 fruits.insert('Apple', 'head');
46 fruits.insert('Banana', 'Apple');
47 fruits.insert('Pear', 'Banana');
48
49 console.log(fruits.display());
50
51 // remove: 从链表中删除一个节点
52 // 查找带删除节点的前一个节点
53 function findPrev(item) {
54     let currentNode = this.head;
55     while (!(currentNode.next == null) && (currentNode.next.element != item)) {
56         currentNode = currentNode.next;
57     }
58     return currentNode;
59 };
60 //删除节点
61 function remove(item) {
62     let prevNode = this.findPrev(item);
63     if (!(prevNode.next == null)) {
64         prevNode.next = prevNode.next.next;
65     }
66 };
67 fruits.insert('Grape', 'Pear');
68 console.log(fruits.display()); // Apple// Banana// Pear// Grape
69
70 fruits.remove('Banana');
71 console.log(fruits.display()); // Apple
72                                 // Pear
73                                 // Grape
74 </script>

```

双向链表

尽管从链表的头节点遍历链表很简单，但是反过来，从后向前遍历却不容易。我们可以通过给Node类增加一个previous属性，让其指向前驱节点的链接，这样就形成了双向链表，如下图：



双向链表

此时，向链表插入一个节点就要更改节点的前驱和后继了，但是删除节点的效率提高了，不再需要寻找待删除节点的前驱节点了。

双向链表的实现

要实现双向链表，首先需要给 Node 类增加一个 previous 属性：

```
1 //节点类
2
3 function Node(element) {
4     this.element = element;    //当前节点的元素
5     this.next = null;          //下一个节点链接
6     this.previous = null;      //上一个节点链接
7 }
```

双向链表的 insert 方法与单链表相似，但需要设置新节点的 previous 属性，使其指向该节点的前驱，定义如下：

```
1 //插入节点
2 function insert ( newElement , item ) {
3     var newNode = new Node( newElement );
4     var currNode = this.find( item );
5     newNode.next = currNode.next;
6     newNode.previous = currNode;
7     currNode.next = newNode;
8 }
```

双向链表的删除 remove 方法比单链表效率高，不需要查找前驱节点，只要找出待删除节点，然后将该节点的前驱 next 属性指向待删除节点的后继，设置该节点后继 previous 属性，指向待删除节点的前驱即可。定义如下：

```
1 //删除节点
2
3 function remove ( item ) {
4     var currNode = this.find ( item );
5     if( !( currNode.next == null ) ){
6         currNode.previous.next = currNode.next;
7         currNode.next.previous = currNode.previous;
8         currNode.next = null;
9         currNode.previous = null;
10    }
```


还有一些反向显示链表 `dispReverse`，查找链表最后一个元素 `findLast` 等方法，相信你已经有了思路，这里我给出一个基本双向链表的完成代码，供大家参考。

```
1  //节点
2
3  function Node(element) {
4      this.element = element;    //当前节点的元素
5      this.next = null;           //下一个节点链接
6      this.previous = null;       //上一个节点链接
7  }
8
9  //链表类
10
11 function LList () {
12     this.head = new Node( 'head' );
13     this.find = find;
14     this.findLast = findLast;
15     this.insert = insert;
16     this.remove = remove;
17     this.display = display;
18     this.dispReverse = dispReverse;
19 }
20
21 //查找元素
22
23 function find ( item ) {
24     var currNode = this.head;
25     while ( currNode.element != item ){
26         currNode = currNode.next;
27     }
28     return currNode;
29 }
30
31 //查找链表中的最后一个元素
32
```

```
33 function findLast () {
34     var currNode = this.head;
35     while ( !( currNode.next == null ) ){
36         currNode = currNode.next;
37     }
38     return currNode;
39 }
40
41
42 //插入节点
43
44 function insert ( newElement , item ) {
45     var newNode = new Node( newElement );
46     var currNode = this.find( item );
47     newNode.next = currNode.next;
48     newNode.previous = currNode;
49     currNode.next = newNode;
50 }
51
52 //显示链表元素
53
54 function display () {
55     var currNode = this.head;
56     while ( !(currNode.next == null) ){
57         console.debug( currNode.next.element );
58         currNode = currNode.next;
59     }
60 }
61
62 //反向显示链表元素
63
64 function dispReverse () {
65     var currNode = this.findLast();
66     while ( !( currNode.previous == null ) ){
67         console.log( currNode.element );
68         currNode = currNode.previous;
69     }
70 }
71
72 //删除节点
```

```

73
74 function remove ( item ) {
75     var currNode = this.find ( item );
76     if( !( currNode.next == null ) ){
77         currNode.previous.next = currNode.next;
78         currNode.next.previous = currNode.previous;
79         currNode.next = null;
80         currNode.previous = null;
81     }
82 }
83
84 var fruits = new LList();
85
86 fruits.insert('Apple' , 'head');
87 fruits.insert('Banana' , 'Apple');
88 fruits.insert('Pear' , 'Banana');
89 fruits.insert('Grape' , 'Pear');
90
91 console.log( fruits.display() );           // Apple
92                                           // Banana
93                                           // Pear
94                                           // Grape
95
96 console.log( fruits.dispReverse() );       // Grape
97                                           // Pear
98                                           // Banana
99                                           // Apple

```

循环链表

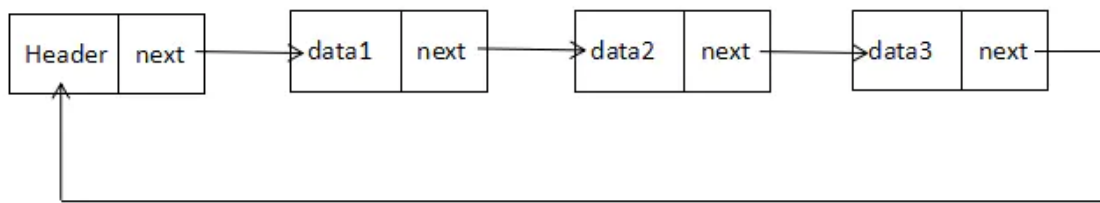
循环链表和单链表相似，节点类型都是一样，唯一的区别是，在创建循环链表的时候，让其头节点的 next 属性执行它本身，即

```

1 head.next = head;

```

这种行为会导致链表中每个节点的 next 属性都指向链表的头节点，换句话说，也就是链表的尾节点指向了头节点，形成了一个循环链表，如下图所示：



循环链表

作者：Cryptic

链接：<https://www.jianshu.com/p/f254ec665e57>

来源：简书

著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。